rt-muse: measuring real-time characteristics of execution platforms

Real-Time Syst
DOI 10.1007/s11241-017-9284-5

`rt-muse`: measuring real-time characteristics of execution platforms

Martina Maggio[1] · **Juri Lelli**[2] · **Enrico Bini**[3]

Abstract Operating systems code is often developed according to principles like simplicity, low overhead, and low memory footprint. Schedulers are no exceptions. A scheduler is usually developed with flexibility in mind, and this restricts the ability to provide real-time guarantees. Moreover, even when schedulers can provide real-time guarantees, it is unlikely that these guarantees are properly quantified using theoretical analysis that carries on to the implementation. To be able to analyze the guarantees offered by operating systems' schedulers, we developed a publicly available tool that analyzes timing properties extracted from the execution of a set of threads and computes the lower and upper bounds to the supply function offered by the execution platform, together with information about migrations and statistics on execution times. `rt-muse` evaluates the impact of many application and platform characteristics including the scheduling algorithm, the amount of available resources, the usage of shared resources, and the memory access overhead. Using `rt-muse`, we show the impact of Linux scheduling classes, shared data and application parallelism, on the delivered computing capacity. The tool provides useful insights on the runtime behavior of the applications and scheduler. In the reported experiments, `rt-muse` detected some issues arising with the real-time Linux scheduler: despite having avail-

✉ Martina Maggio
martina@control.lth.se

Juri Lelli
juri.lelli@arm.com

Enrico Bini
bini@di.unito.it

1 Department of Automatic Control, Lund University, Lund, Sweden

2 ARM, Cambridge, UK

3 University of Turin, Turin, Italy

Published online: 30 June 2017

able cores, Linux does not migrate SCHED_RR threads which are enqueued behind SCHED_FIFO threads with the same priority.

Keywords Supply functions · Resource measurement · Linux kernel · Tools

1 Introduction

Real-time resource allocation policies are usually developed together with theoretical results that support claims about their behavior. However, these claims apply to ideal conditions and it is unknown if they hold when the policies are implemented in real operating systems (OSes). For this reason, these results are often confined to theory. OS developers are resistant to the adoption of real-time algorithms, mostly because general purpose OSes are designed for flexibility and to operate in complex and variable scenarios compared to the one assumed in real-time systems.

The gap between theory and practice can be closed by making the implementation of real-time policies more accessible to non-kernel experts. A notable example in this direction is LITMUS^{RT} (Calandrino et al. 2006). LITMUS^{RT} offers a simple environment to implement sophisticated scheduling policies with a reasonably low effort. PD^2 (Anderson and Srinivasan 2001) and RUN (Regnier et al. 2011) are just two examples of scheduling policies implemented in LITMUS^{RT}.

Another way to fill the gap is to determine real-time characteristics of existing schedulers' implementations. `rt-muse` aims to go in this second direction. The timing properties of scheduling policies can indeed be monitored. Feather-Trace (Brandenburg and Anderson 2007), trace-cmd,[1] and kernelshark[2] are just examples of tools, which help to monitor and to visualize some properties of the execution of threads. However, the analysis of the large amount of data produced by these tracing tools often needs to be extracted and interpreted by experts. `rt-muse` is an easy-to-use tool to extract the real-time characteristics of an execution platform in an interpretable way.

Contribution `rt-muse` measures real-time characteristics of platforms by running experiments with a synthetic taskset with user-defined characteristics and extracting sequences of timestamps from the run. These sequences are then analyzed for real-time features like supply functions and the number of migrations, together with statistics on the response times of jobs. The analysis performed by `rt-muse` is organized in modules, making the tool easily extensible. The advantage of a modular analysis is that the user can specify the analysis of interest for any thread or for the taskset. A modular analysis allows creating new modules easily. In `rt-muse`, implementing a new module requires the creation of a Matlab/Octave function to perform the desired operations. This paper extends the previous contribution of `rt-muse` (Maggio et al. 2016) adding a new analysis module, the statistical module presented in Sect. 3.4.

This paper also presents a broad set of experiments, covering several scheduling policies and settings. Some of the experiments are particularly insightful and sug-

[1] https://lwn.net/Articles/410200/.

[2] http://rostedt.homelinux.com/kernelshark/.

gest that the current implementation of the Linux scheduling classes may actually be improved. For example, the experiment described in Sect. 6.3 shows that threads that could migrate and execute do not receive any CPU because of a problem with the push/pull migration. As a new contribution of this paper over the previous publication (Maggio et al. 2016), two new sets of experiments are introduced, respectively in Sects. 6.2 and 6.5, to further demonstrate the results that the tool allows one to obtain. In particular, the experiment presented in Sect. 6.2 describes the behavior of `SCHED_OTHER` and shows that it cannot be used as a real-time scheduling policy, since it does not provide any guarantee on the computation time allocated to the threads in the taskset and the set of experiments presented in Sect. 6.5 analyze the `SCHED_DEADLINE` scheduling class both from the perspective of how it manages a single reservation and in the presence of an additional load.

The results of the experiments performed by `rt-muse` are reproducible, as we rely on tools that are integrated into the Linux kernel.

2 Overview of `rt-muse`

The goal of `rt-muse` is to measure the real-time characteristics of execution platforms (Sect. 3 describes precisely the real-time characteristics extracted by `rt-muse`). The platform includes the hardware, the operating system and all of its components, among which the scheduler. `rt-muse` achieves its goal by analyzing the traces of the execution of a set of experiments, properly constructed in accordance to the user's specifications.

The steps for using `rt-muse` are illustrated in Fig. 1. The first step is the creation of a set-up file (in JSON format) that specifies the characteristics of the platform that we want to extract. The format of the configuration file is described in Sect. 5.

To avoid the bias introduced by the measuring infrastructure, the target machine that runs the experiments and the host machine collecting and analyzing the execution traces are different.

Once the configuration file is completed, the invocation of a script file at the host machine takes care of:

- Copying the configuration file to the target machine;
- starting the experiments at the target machine. While the experiment is in progress, the target machine sends timestamps (via UDP datagrams) to the host machine that collects them;
- as soon as the experiment terminates, the target machine informs the host machine, which analyzes the data collected in accordance to the measurements requested by the user.

During run-time, the timestamping infrastructure records the instant $t_{i,j}$ when the j-th execution of the job body belonging to the thread τ_i starts and the processing unit index $\pi_{i,j}$. To record these timestamps, as well as to monitor other kernel events (thread migrations, scheduler invocation, etc.), we use `trace-cmd`,[3] the user-space

[3] http://lwn.net/Articles/410200/.

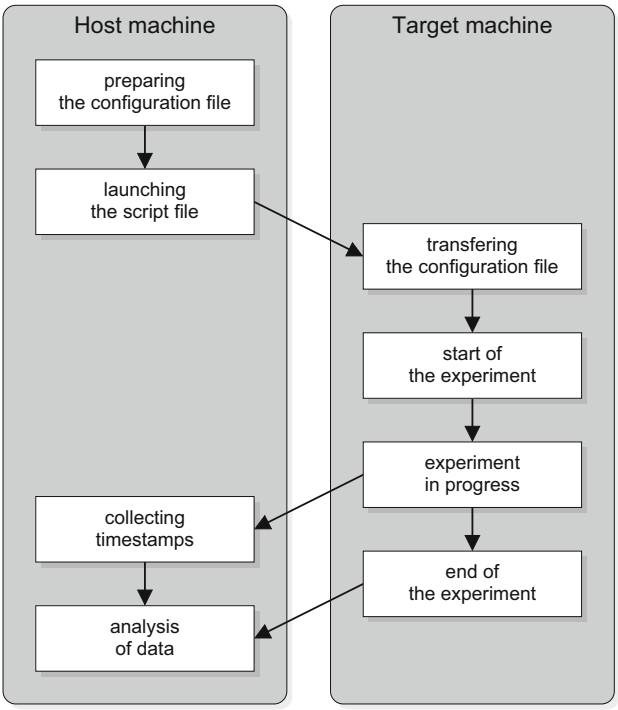


Fig. 1 Steps of `rt-muse` measurement procedure

front end for Ftrace.[4] Ftrace is a lightweight tracer of kernel events developed by Steven Rostedt, one of the Linux kernel developers. Together with kernel events, such as thread migrations or context switches, it can monitor user-defined events. On x86 machines, Ftrace has nanoseconds precision.

Events can be collected by Ftrace in two ways: (1) stored in a buffer, which is then flushed to disk when full, or (2) sent via User Datagram Protocol (UDP) to a listener instance of Ftrace running on a remote machine. Due to disk writing, if the collector is executed on the same machine, there is a risk of service outage. This is especially true if a large number of events is generated. To avoid this problem and to minimize the impact of the tool itself on the monitored experiment, we used the remote monitoring via UDP. The drawback of using UDP is the possibility of losing some events. Although this never happened in the experiments reported in the paper, `rt-muse` is capable to reconstruct any missing event, by linearly interpolating the two neighboring ones.

`rt-muse` is invoked on a host machine and performs the following steps:

1. It sends the experiment description, formatted as described in Sect. 4, to the target machine, which is the one actually running the experiment;
2. It starts listening events sent by the target machine;

[4] http://elinux.org/Ftrace.

3. It communicates to the target machine to start the experiment run;
4. It collects all received events until the target communicates the completion of the experiment;
5. It performs all the analysis modules requested by the user.

After the script file on the host machine has terminated, the user can read the output of the experiment in a human readable form.

Finally, `rt-muse` is freely available on-line.[5]

Organization of the paper In the following, Sect. 3 describes the analysis modules available. We start with the analysis to highlight what are the benefits of analyzing a taskset execution trace with `rt-muse`. The paper then describes the application model and how it is possible to construct a synthetic taskset that models the real application in Sect. 4. `rt-muse` focuses on one specific type of applications (applications that repeats the same job continuously, like multimedia applications, streaming services, image analysis software and many others). Sections 5 and 6 then describe how the tool can be used, and the results of the experiments that we have conducted with `rt-muse`. Finally, Sect. 7 discusses related research efforts and Sect. 8 concludes the paper.

3 The analysis capabilities of `rt-muse`

This section describes the analysis modules offered by `rt-muse`. Each of the following subsections focuses on one of specific module currently implemented. `rt-muse` is engineered to make it simple to add a new analysis module and to use the data produced by other modules for further investigations.

First, Sect. 3.1 describes the supply analysis module and the additional background and definitions necessary to understand the analysis method. The supply module computes an experiment-based approximation of the lower and upper supply bound functions of the computing resource allocated to a single thread as well as the bounds to the overall resource allocated to a set of threads. The section also lists several properties, which are exploited for computing bounds to the supply functions. Second, in Sect. 3.2 we describe the runmap module, which computes a map with information about where threads are executed. Third, in Sect. 3.3 we introduce the migration module, that computes the migrations experiences by each thread. Finally, in Sect. 3.4 we introduce the statistical module that computes statistics about the average response time of jobs.

3.1 The supply analysis module

The supply analysis module aims at extracting an experiment-based supply function from the trace of execution of the threads in the taskset. For each of the threads we want to approximate the computational capacity offered by the platform with a powerful analysis tool. Supply functions have been used, possibly under different names in different contexts [e.g. *service curves* in network/real-time calculus (Cruz

[5] https://github.com/martinamaggio/rt-muse.

1991; Baccelli et al. 1992; Thiele et al. 2000)], to model the availability of different types of resource: processing time of a single (Mok et al. 2001; Lipari and Bini 2003; Shin and Lee 2003) or multi-core (Bini 2009; Xu et al. 2015) platform, network (Cruz 1991; Almeida et al. 2002), memory (Yun et al. 2016; Åkesson and Goossens 2011), and I/O (Pellizzoni et al. 2008). However, this is the first instance in which a supply function is constructed based on the experimental execution trace.

To understand how supply functions are extracted from the application execution, let us first define an *execution platform* $\mathcal{P}$ as any mechanism that provides computing capacity [such as a virtual machine, a priority level of a scheduler, or a periodic resource (Lipari and Bini 2003; Shin and Lee 2003)]. The supply analysis module computes the *experiment-based supply lower* (respectively *upper*) *bound functions* $\widetilde{\mathsf{slbf}}(t)$ (resp. $\widetilde{\mathsf{subf}}(t)$) of the platform $\mathcal{P}$, based on actual measurements.

The instrument of the supply module to measure the supply functions of a platform $\mathcal{P}$, is a set $\mathcal{T} = \{\tau_1, \ldots, \tau_n\}$ of n threads which are going to be executed over the platform itself. Each of the n threads $\tau_i \in \mathcal{T}$ executes forever the same *job* in loop, with nominal duration e_i. The job body can be configured by the user, as described in Sect. 4, to test the response of the platform to different types of loads. For the duration of the experiment, the start time $t_{i,j}$ of the j-th job of the i-th thread is recorded by the monitoring infrastructure. The supply module extracts the supply lower and upper bounds by comparing the length of sequences of jobs with their nominal value. Intuitively, the longer the jobs take, the smaller is the amount of computing capacity delivered by the platform $\mathcal{P}$.

In the following we present our implementation choices for the supply module. Before, we recall some background on supply functions. The resource allocation operated by the scheduler to thread τ_i is modeled by means of a *scheduling function* $\xi_i(t)$ with the standard interpretation

$$\xi_i(t) = \begin{cases} 1 & \tau_i \text{ runs at } t \\ 0 & \text{otherwise.} \end{cases} \tag{1}$$

The *supply lower bound function* $\mathsf{slbf}_i(t)$ and the *supply upper bound function* $\mathsf{subf}_i(t)$ of thread τ_i are functions such that (Mok et al. 2001; Lipari and Bini 2003; Shin and Lee 2003)

$$\mathsf{slbf}_i(b-a) \leq \int_a^b \xi_i(t)\,dt \leq \mathsf{subf}_i(b-a), \tag{2}$$

meaning that $\mathsf{slbf}_i(t)$ and $\mathsf{subf}_i(t)$ are lower and upper bounds to the amount of resource allocated to the thread τ_i in any interval of length t.

Supply functions also determine bounds to the completion time $R(W)$ of any amount of work W to be made by the thread τ_i. In fact

$$R_{\mathsf{b}}(W) \leq R(W) \leq R_{\mathsf{w}}(W), \tag{3}$$

with $R_{\mathsf{b}}(W)$ and $R_{\mathsf{w}}(W)$ defined as

$$\begin{cases} R_{\mathsf{w}}(W) = \sup\{t : \mathsf{slbf}_i(t) < W\} \\ R_{\mathsf{b}}(W) = \inf\{t : \mathsf{subf}_i(t) \geq W\}. \end{cases} \tag{4}$$

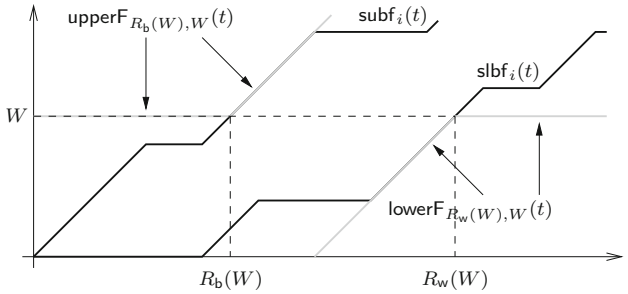


Fig. 2 The completion time of an amount of work W can be bounded by $R_{\mathsf{b}}(W)$ and $R_{\mathsf{w}}(W)$, which are determined by the supply bound functions $\mathsf{subf}_i(t)$ and $\mathsf{subf}_i(t)$, respectively, from Eq. (4). The construction of the supply bound functions via Corollary 1 is also illustrated

Figure 2 illustrates the bounds to the completion time $R(W)$ of (4), as function of the pair $\big(\mathsf{subf}_i(t), \mathsf{slbf}_i(t)\big)$.

From (1), it follows that the *maximum instantaneous execution rate* is of any thread is $\alpha_i^{\mathsf{max}} = 1$. Supply functions are Lipschitz-continuous with constant α_i^{max}, that is

$$\frac{\partial\, \mathsf{slbf}_i(t)}{\partial t}, \frac{\partial\, \mathsf{subf}_i(t)}{\partial t} \in [0, \alpha_i^{\mathsf{max}}]. \tag{5}$$

Despite the value of α_i^{max} being 1, we keep using the explicit notation α_i^{max}, since later in Sect. 3.1 we are going to analyze also the case of supply functions of a set of threads possibly running in parallel, hence with a maximum instantaneous execution rate higher than 1.

The determination of the exact supply functions requires the exploration of all possible resource schedules, which is unrealistic over a real operating system. Instead, we propose to infer the experiment-based supply function by measuring the bounds $R_{\mathsf{w}}(W)$ and $R_{\mathsf{b}}(W)$ of the completion time of a known amount of work W. This is made possible by the next Theorem.

Theorem 1 *Let $R_{\mathsf{w}}(W)$ and $R_{\mathsf{b}}(W)$ be respectively the largest and smallest completion times of an amount W of work over a platform with maximum instantaneous rate α_i^{max}.*

Then, the supply lower bound function $\mathsf{slbf}_i(t)$ is bounded from below by

$$\mathsf{slbf}_i(t) \geq \begin{cases} W + \alpha_i^{\mathsf{max}} \times (t - R_{\mathsf{w}}(W)) & t \leq R_{\mathsf{w}}(W) \\ W & t > R_{\mathsf{w}}(W), \end{cases} \tag{6}$$

and the supply upper bound function $\mathsf{subf}_i(t)$ is bounded from above by

$$\mathsf{subf}_i(t) \leq \begin{cases} W & t < R_{\mathsf{b}}(W) \\ W + \alpha_i^{\mathsf{max}} \times (t - R_{\mathsf{b}}(W)) & t \geq R_{\mathsf{b}}(W). \end{cases} \tag{7}$$

Proof First, we prove the Theorem for the $\mathsf{slbf}_i(t)$. By the definition of $R_{\mathsf{w}}(W)$ in (4), it follows that

$$t > R_{\mathsf{w}}(W) \quad \Rightarrow \quad \mathsf{slbf}_i(t) \geq W. \tag{8}$$

In fact, if there is any $t^* > R_{\mathsf{w}}(W)$ with $\mathsf{slbf}_i(t^*) \geq W$, then $R_{\mathsf{w}}(W)$ would be at least as big as t^*, which is a contradiction.

For any $t \leq R_{\mathsf{w}}(W)$ we invoke the Lipschitz-continuity of $\mathsf{slbf}_i(t)$ with constant α_i^{max}, that is

$$\begin{aligned} \mathsf{slbf}_i(R_{\mathsf{w}}(W)) - \mathsf{slbf}_i(t) &\leq \alpha_i^{\mathsf{max}}(R_{\mathsf{w}}(W) - t) \\ \mathsf{slbf}_i(t) &\geq \mathsf{slbf}_i(R_{\mathsf{w}}(W)) + \alpha_i^{\mathsf{max}}(t - R_{\mathsf{w}}(W)). \end{aligned}$$

From (4), since $R_{\mathsf{w}}(W)$ is the supremum of a set, then there is a sequence $\{t_k\}_{k\in\mathbb{N}}$ within $\{t : \mathsf{slbf}_i(t) < W\}$ such that $\lim_k t_k = R_{\mathsf{w}}(W)$ and $\lim_k \mathsf{slbf}_i(t_k) = W$. From the continuity of $\mathsf{slbf}_i(t)$, it follows that $\mathsf{slbf}_i(R_{\mathsf{w}}(W)) = W$ and then from the Lipschitz-continuity

$$t \leq R_{\mathsf{w}}(W) \quad \Rightarrow \quad \mathsf{slbf}_i(t) \geq W + \alpha_i^{\mathsf{max}}(t - R_{\mathsf{w}}(W)), \tag{9}$$

which proves, together with (8), the lower bound on $\mathsf{slbf}_i(t)$ of (6). The proof of (7) is analogous and it omitted for brevity. □

The inequalities (6) and (7), can be written more compactly as

$$\begin{aligned} &\forall t, \quad \mathsf{slbf}(t) \geq \mathsf{lowerF}_{R_{\mathsf{w}}(W),W}(t) \\ &\forall t, \quad \mathsf{slbf}(t) \leq \mathsf{upperF}_{R_{\mathsf{b}}(W),W}(t) \end{aligned}$$

respectively, with the auxiliary functions $\mathsf{lowerF}_{x,y}(t)$ and $\mathsf{upperF}_{x,y}(t)$ illustrated in 2, and defined as

$$\mathsf{lowerF}_{x,y}(t) = \begin{cases} y + \alpha^{\mathsf{max}} \times (t - x) & t \leq x \\ y & t > x, \end{cases} \tag{10}$$

$$\mathsf{upperF}_{x,y}(t) = \begin{cases} y & t < x \\ y + \alpha^{\mathsf{max}} \times (t - x) & t \geq x. \end{cases} \tag{11}$$

Since the bounds of (6) and (7) hold for any amount of work W, the next corollary follows.

Corollary 1 *Let*

- $\mathsf{lowerF}_{x,y}(t)$ *and* $\mathsf{upperF}_{x,y}(t)$ *be defined as in (10) and (11), respectively, and*
- $R_{\mathsf{w}}(W)$ *and* $R_{\mathsf{b}}(W)$ *be longest and shortest completion time of an amount of work* W *over a platform with maximum instantaneous execution rate* α_i^{max}.

Then the following bounds on the $\mathsf{slbf}_i(t)$ *and* $\mathsf{subf}_i(t)$ *hold*

$$\mathsf{slbf}_i(t) \geq \sup_{W \geq 0} \left\{ \mathsf{lowerF}_{R_{\mathsf{w}}(W),W}(t) \right\} \tag{12}$$

$$\mathsf{subf}_i(t) \leq \inf_{W \geq 0} \left\{ \mathsf{upperF}_{R_{\mathsf{b}}(W),W}(t) \right\}. \tag{13}$$

Equations (12), (13) provide an alternate definition of the bounds to supply functions. The advantage of (12), (13) over the standard definitions of supply functions is that they depend on the bounds $R_{\mathsf{w}}(W)$ and $R_{\mathsf{b}}(W)$ of the completion time of the work W made by thread τ_i. These values, in fact, can be measured by targeted experiments and used, through (12) and (13), to compute the experiment-based supply functions. More precisely, the steps made by `rt-muse` to measure the supply functions of a platform are

1. the set of threads $\{\tau_1, \ldots, \tau_n\}$, each one with nominal duration e_i, are created by the user (Sect. 4 describes how);
2. from the execution traces, the sequence of timestamps $t_{i,j}$ of the start time of the j-th job of τ_i is extracted (j from 0 to the index maxJ_i of the latest timestamp);
3. from the sequence of timestamps $\{t_{i,j}\}_{j=0}^{\mathsf{maxJ}_i}$, following sequences

$$\forall k = 0, \ldots, \mathsf{maxJ}_i \qquad \overline{s}_{i,k} = \max_{j=0,\ldots,\mathsf{maxJ}_i - k} \{t_{i,j+k} - t_{i,j}\}$$

$$\forall k = 0, \ldots, \mathsf{maxJ}_i \qquad \underline{s}_{i,k} = \min_{j=0,\ldots,\mathsf{maxJ}_i - k} \{t_{i,j+k} - t_{i,j}\},$$

   are computed (notice, it is always $\overline{s}_{i,0} = \underline{s}_{i,0} = 0$);
4. since each τ_i is structured as forever loop, we approximated the longest and shortest time to complete an amount of work equal to $k \times e_i$ as follows

$$R_{\mathsf{w}}(k \times e_i) \approx \overline{s}_{i,k} \qquad R_{\mathsf{b}}(k \times e_i) \approx \underline{s}_{i,k}. \tag{14}$$

5. finally, in accordance to the bounds of (12) and (13), we define the *experiment-based supply upper/lower bound functions* as

$$\widetilde{\mathsf{slbf}}_i(t) = \max_{k=0\ldots,\mathsf{maxJ}_i} \left\{ \mathsf{lowerF}_{\overline{s}_{i,k},ke_i}(t) \right\} \tag{15}$$

$$\widetilde{\mathsf{subf}}_i(t) = \min_{k=0\ldots,\mathsf{maxJ}_i} \left\{ \mathsf{upperF}_{\underline{s}_{i,k},ke_i}(t) \right\}. \tag{16}$$

For thread τ_i, the job *nominal length* e_i represents the duration of the job body in ideal conditions (with no interference by the execution of other threads). In our experiment-based computation, this value is set to

$$e_i = \underline{s}_{i,1}, \tag{17}$$

that is the shortest duration among all job of τ_i in the experiment. The user can also specify any other value. Since the job lengths extracted from the timestamps sequence

have to be compared against the nominal length e_i, setting e_i to a bigger value may lead to too optimistic conclusions about the computing capacity of the platform, while setting e_i smaller than $\underline{s}_{i,1}$ may be too conservative.

To summarize the content of supply functions in a more compact and understandable form, they are often abstracted by the bandwidth and the delay. More precisely, with a given $\widetilde{\mathsf{slbf}}_i(t)$, we say that the supply lower bound function has *bandwidth* $\underline{\alpha}_i$ and *delay* $\underline{\Delta}_i$ if

$$\forall t, \quad \widetilde{\mathsf{slbf}}_i(t) \geq \underline{\alpha}_i(t - \underline{\Delta}_i). \tag{18}$$

Among the many pairs $(\underline{\alpha}, \underline{\Delta})$ satisfying (18), we choose the one such that the corresponding linear lower bound better approximate the exact supply lower bound function $\mathsf{slbf}(t)$. Such a pair corresponds to the solution of the following maximization problem

$$\begin{aligned} &\text{maximize} \quad \int_{\underline{\Delta}}^{H} \underline{\alpha}(t - \underline{\Delta}) \\ &\text{such that} \quad \mathsf{slbf}(t) \geq \underline{\alpha}(t - \underline{\Delta}) \quad \forall t \end{aligned}$$

with H being a user-defined time-horizon of interest. In fact, the area below the linear lower bound over the interval of interest $[0, H]$ can be considered a measure of the tightness of the linear lower bound.

Analogously as in (18), we also define the bandwidth and delay of the supply upper bound function as any pair $(\overline{\alpha}_i, \overline{\Delta}_i)$ such that:

$$\forall t, \quad \widetilde{\mathsf{subf}}_i(t) \leq \overline{\alpha}_i(t - \overline{\Delta}_i). \tag{19}$$

Notice that from the property that $\widetilde{\mathsf{subf}}_i(0) = 0$, it follows that $\overline{\Delta}_i$ is always going to be non-positive. Similarly to the lower bound case, to among all linear upper bounds satisfying (19), our tool computes explicitly the pair $(\overline{\alpha}_i, \overline{\Delta}_i)$ among the feasible ones, such that the area below the linear upper bound over the interval $[0, H]$ is minimized. To enable the computation of different linear upper/lower bounds, the tool also computes the convex hull of both the lower and the upper bound functions.

The supply module also allows the characterization of the overall computing capacity delivered by the platform $\mathcal{P}$ to the entire set of n threads $\mathcal{T}$. These measures are denoted by the subscript "$*$" rather than "i" of the per-thread analysis. The steps are exactly the same as for the per-thread analysis, with the following adaptation

1. the overall schedule is defined by

$$\xi_*(t) = \sum_{\tau_i \in \mathcal{T}} \xi_i(t),$$

2. the maximum instantaneous execution rate is set to $\alpha_*^{\max} = \min(n, m)$, with m being the number of processors of the platform, and

3. the timestamps are

$$\{t_{*,k}\} = \bigcup_{i=1}^{n} \{t_{i,k}\},$$

since, in this case, we aim at measuring the amount of computation delivered by the platform to any thread.

After the analysis of the overall platform $\mathcal{P}$ is completed, the results are the supply function bounds $\widetilde{\mathsf{slbf}}_*(t)$, $\widetilde{\mathsf{subf}}_*(t)$, the bandwidth/delay pair of the linear lower bound $(\underline{\alpha}_*, \underline{\Delta}_*)$, and the pair of the linear upper bound $(\overline{\alpha}_*, \overline{\Delta}_*)$.

3.2 The runmap analysis module

At the begin of all jobs, Ftrace collects both the timestamp $t_{i,j}$ as well as the index of the CPU on which the j-th job by the i-th thread started on, denoted with $\pi_{i,j}$. This information is used by the runmap analysis module, together with the total number of jobs terminated by each thread, to compute the runmap. For each CPU included in the affinity mask of the thread, the tool reports the percentage of jobs that started on that CPU. This analysis can only be performed per thread.

3.3 The migration analysis module

For each thread, we compute the number of migrations experienced by the thread during its execution. This analysis can only be performed per thread. The migration module analyzes migrations data and allows one to compute the frequency of migration over time. This will make the information about conditions in which thread are constantly migrated more accessible to the user, who can then understand when the limits of the platform have been reached.

3.4 The statistical analysis module

If we define the random variable $s_{i,k}$ as the time it takes to execute k consecutive jobs of the thread τ_i (i.e., the time elapsing between the start times of $k+1$ consecutive jobs), we can say that the supply analysis module provides information about the two random variables $\min\{s_{i,k}\}$ and $\max\{s_{i,k}\}$. The statistical analysis module, instead, investigates the mean and the variance of the random variable $s_{i,k}$. If we assume that the timestamps of the start times of τ_i's jobs are indexed from 0 to maxJ_i, then the statistical analysis module computes, for all $k = 1, \ldots, \mathsf{maxJ}_i$, the average $\mu_{i,k}$ and the variance $\sigma^2_{i,k}$, as described below

$$\mu_{i,k} = \frac{\sum_{j=0}^{\mathsf{maxJ}_i - k} (t_{i,j+k} - t_{i,j})}{\mathsf{maxJ}_i - k + 1} \tag{20}$$

$$\sigma_{i,k}^2 = \frac{\sum_{j=0}^{\mathsf{maxJ}_i - k} (\mu_{i,k} - (t_{i,j+k} - t_{i,j}))^2}{\mathsf{maxJ}_i - k + 1} \tag{21}$$

therefore avoiding the last $k-1$ jobs, that would not contribute to a meaningful estimate of the time it takes to execute k jobs.

4 Modeling a taskset with `rt-muse`

The goal of `rt-muse` is to measure the computing capacity delivered by the execution platform to different types of workloads. Such a feature is realized by allowing the testing threads $\mathcal{T}$ to be configured by the user. For this purpose, the job body of each thread τ_i is composed by the sequential concatenation of p_i *job phases* denoted by $\phi_{i,1}, \phi_{i,2}, \ldots, \phi_{i,p_i}$. Each phase $\phi_{i,k}$ belongs to a set Φ of available phases. Currently, the tool offers the following set of phases $\Phi = \{\phi^{\mathsf{compute}}, \phi^{\mathsf{lock}}, \phi^{\mathsf{memory}}, \phi^{\mathsf{shared}}, \phi^{\mathsf{user}}\}$. Each phase may be configured through some input parameters, as described below.

The phase $\phi^{\mathsf{compute}}(\ell)$, called *compute* phase, is configured with a positive integer ℓ. The execution of the phase $\phi^{\mathsf{compute}}(\ell)$ keeps the CPU busy for a time proportional to ℓ. In our current implementation, this phase executes ℓ floating point operations. This can be used to simulate any computational intensive work such as the run of a control algorithm, the aggregation of data from different sensors or else.

The phase $\phi^{\mathsf{lock}}(\ell, r)$, called *lock* phase, is configured with a positive integer ℓ and an index r of a shared resource. The body of the phase is exactly the same as the compute phase. However, before starting the execution of the mathematical operations, the thread locks the shared resource r. The lock is released at the end of the execution. Resource identifiers are numbered from 0 to $R-1$, where R is the total number of available resources, as specified in the test configuration file. We use the default `pthread_mutex_lock` function offered by glibc version 2.19. This phase can, therefore, be used to evaluate the impact of shared resources on the overall computing capacity and to test different locking protocols.

The phase $\phi^{\mathsf{memory}}(\ell, m)$, called *memory* phase, is configured with a positive integer ℓ and a size m of memory to be used. The body of the phase is exactly the same as the compute phase. However, before entering the computation part, the phase dynamically allocates an amount of memory corresponding to m double values on the heap and saves the results of the mathematical computation in the allocated vector. Whenever the result storage reaches the end of the vector, it restarts from the beginning. This phase can be used to test the platform in presence of memory writing and to evaluate the impact of cache sizes on the performance delivered to the running applications.

The phase $\phi^{\mathsf{shared}}(\ell)$, called *shared* phase, is configured with a positive integer ℓ. The phase behaves similarly to the memory phase, but it writes the data to a memory buffer shared among all the threads (rather than in a private buffer as in ϕ^{memory}). The shared memory is locked before usage and the lock is released when the ℓ operations are completed. The size of the shared memory buffer is specified as a configuration parameter for the entire application and the memory is allocated on the heap at start-up.

This phase can be used to evaluate the performance degradation in presence of shared memory.

Finally, the phase ϕ^{user} offers the user the possibility to define any user-specific phase. During this phase, the user function is invoked. The code of this phase can be modified by the user to analyze the computing capacity delivered by the platform to any type of application, which cannot be precisely described by any sequence of the predefined phases.

5 Specification of experiments

As anticipated in Sect. 2, an experiment is defined by a configuration file that specifies the taskset characteristics and the execution platform characteristics. This file is formatted using the JavaScript Object Notation (JSON[6]). As illustrated in the example of Listing 1, and has four main sections ("objects" in the JSON terminology):

1. `global`, specifying the global settings,
2. `resources`, specifying the available resources,
3. `shared`, denoting the shared memory size, and
4. `threads`, describing the threads in $\mathcal{T}$ to be executed.

The `global` configuration section (lines 1–4 in the example of Listing 1) allows the user to specify the default scheduling policy to be used. Currently, only Linux machines are allowed, and the scheduling classes can be chosen among `SCHED_OTHER`, `SCHED_RR`, `SCHED_FIFO` and `SCHED_DEADLINE`, which are implemented in the latest kernel (at line 2, `SCHED_RR` is specified as default). Also, the duration of the experiment in seconds is specified in the `global` section (10 seconds in the example, at line 3). The `resources` section (line 6 in the example) reports the number R of total different shared resources to be used by any ϕ^{lock} phase. In the lock phase, one can specify that the thread locks the resource $r = 0, \ldots, R-1$. The section `shared` (line 7 in the example) specifies the size in bytes of the shared memory buffer to be shared by the ϕ^{shared} thread phases. Such a buffer is protected with an additional lock (not belonging to the R mutexes specified in the `resources` section).

```
1  { "global" : {
2      "default_policy" : "SCHED_RR",
3      "duration" : 10,
4      "analysis": { "supply": true }
5    },
6    "resources": 2,
7    "shared": 50,
8    "threads": {
9      "thread1": {
10       "priority": 10,
11       "cpus": [0,1,2],
12       "phases": {
13         "c0": { "loops": 1000 },
14         "l0": { "loops": 2000, "res": 0 },
15         "s0": { "loops": 1000 },
```

[6] http://www.json.org.

```
16          "l1": { "loops": 3000, "res": 1 },
17          "m0": { "loops": 1000, "memory": 10 },
18          "c1": { "loops": 5000 }
19        },
20        "analysis": { "supply": true,
21                      "runmap": true,
22                      "migrations": true,
23                      "statistical": true }
24      },
25      "thread2": {
26        "cpus": [0,1],
27        "policy": "SCHED_DEADLINE",
28        "budget": 1000,
29        "period": 2000,
30        "phases": {
31          "c0": { "loops": 5000 },
32          "m0": { "loops": 1000, "memory": 100 },
33          "l0": { "loops": 10000, "res": 1 }
34  } } } }
```

Listing 1 Example of JSON test configuration file.

The threads $\tau \in \mathcal{T}$ are described in the `thread` section (lines 8–34 in the example). For each thread τ_i the scheduling parameters are specified. These parameters are used by the scheduler to select the threads to be executed over the execution platform $\mathcal{P}$ (in the example, τ_1 is scheduled by the default scheduling policy `SCHED_RR` with priority 10 set at line 10, while at lines 27–29 τ_2 is set to be scheduled by `SCHED_DEADLINE` with budget 1000 and period 2000 nanoseconds). Also, for each thread it is possible to specify its affinity, that is the subset of the available CPUs on which the thread is allowed to execute (in the example, τ_1 can execute only on CPUs 0, 1, and 2, while τ_2 can execute only on CPUs 0 and 1). If left unspecified, the thread can migrate over all the available CPUs.

Finally, for each thread τ_i the `phases` section contains the list of the p_i phases to be executed in sequence. Each phase has a name and the corresponding parameters. The name of the phase identifies its type. A phase name that starts with the letter `c` is a $\phi^{\mathsf{compute}}(\ell)$ phase and expects to receive the parameter ℓ. A phase name starting with the letter `l` indicates a lock phase $\phi^{\mathsf{lock}}(\ell, r)$ and should receive two parameters, the number of iterations ℓ and the resource r to be locked. A phase name starting with the letter `m` denotes a memory phase $\phi^{\mathsf{memory}}(\ell, m)$ and takes two parameters, the number of iterations ℓ and the amount of memory to be used m. Finally, a phase name starting with the letter `s` identify a shared phase $\phi^{\mathsf{shared}}(\ell)$ and receives one parameter, the number of iterations ℓ. Phases of the same type may repeat within the thread body. In the example, τ_1 has $p_1 = 5$ phases described in lines 12–19. According to the specification of Listing 1, the phases of τ_1 are:

1. $\phi_{1,1} = \phi^{\mathsf{compute}}(1000)$, that is a loop of 1000 mathematical operations,
2. $\phi_{1,2} = \phi^{\mathsf{lock}}(2000, 0)$, that is a loop of 2000 mathematical operations while locking resource 0,
3. $\phi_{1,3} = \phi^{\mathsf{shared}}(1000)$, that is it computes 1000 operations on the shared memory buffer of size 50 (as specified at line 7),
4. $\phi_{1,4} = \phi^{\mathsf{lock}}(1000, 1)$, that is τ_1 uses resource 1, and then does 1000 iterations,

5. $\phi_{1,5} = \phi^{\mathsf{memory}}(1000, 10)$, that is τ_1 does 1000 iterations writing circularly in a memory area of 10 elements, and finally
6. $\phi_{1,6} = \phi^{\mathsf{compute}}(5000)$, that is a loop of 5000 mathematical operations.

Thread τ_2 of the example, instead, has the following phases: $\phi_{2,1} = \phi^{\mathsf{compute}}(5000)$, $\phi_{2,2} = \phi^{\mathsf{memory}}(1000, 100)$, and $\phi_{2,3} = \phi^{\mathsf{lock}}(10000, 1)$.

The `analysis` section can be specified within each thread specification as well as within the `global` section. In both cases, it lists the name of the analysis modules to be executed. In case the `analysis` element is missing (such as for τ_2 in the example), then the corresponding thread data is not analyzed, although executed. This feature can be used to create some workload interfering with the threads to be monitored. When the `analysis` section is present in the `global` section, the corresponding analysis modules are performed over the data aggregated from all threads that contain an `analysis` section. Such a type of aggregated analysis can be used to determine the overall characteristics of the platform (such as the overall delivered computing capacity).

5.1 Example of results

To better illustrate the output of `rt-muse`, we present an example with 6 threads all equal to each other. We assigned them the `SCHED_OTHER` scheduling class and set an affinity mask that contains CPUs [1, 2, 3], meaning that all the 6 threads were confined to run over 3 among the 4 CPUs available on our platform. All thread bodies were composed by the only one phase $\phi_{i,1} = \phi^{\mathsf{compute}}(100000)$, for all $i = 1, \ldots, 6$. The experiment lasted 100 seconds.

The actual duration of the monitored window was 99.0567 s. Compared to the duration of the experiment (set to 100 s), it means that it took about one second to start up the tool, create the threads, take first scheduling decisions, etc. before the first job of all 6 threads was activated.

Let us illustrate the results of the analysis of thread τ_3 (any other selection would give similar data, as all threads are the same). Thread τ_3 released 35838 (denoted by maxJ_3 in Sect. 3.1) jobs in the analysis window. The number of job migrations measured by the `migrations` module is 220. Thread τ_3 did run over the 3 CPUs according to the shares [0.009, 0.440, 0.551]. The experiment-based supply lower/upper bounds $\widetilde{\mathsf{slbf}}_3(t)$ and $\widetilde{\mathsf{subf}}_3(t)$ are plotted in Fig. 3. Notice that, to illustrate the information over different timescales, the lower-left corner of the top figure is properly magnified below. Lower and upper linear bounds, with parameters $(\underline{\alpha}_3, \underline{\Delta}_3)$ and $(\overline{\alpha}_3, \overline{\Delta}_3)$, are also drawn in dashed lines. Finally, we observe that any valid linear bound, other than the ones explicitly computed by our method, can be determined by evaluating only the points over the convex hull of $\widetilde{\mathsf{slbf}}_3(t)$ (or $\widetilde{\mathsf{subf}}_3(t)$) provided with the results. The vertices of the supply function along the convex hull are also drawn in the figure.

For the same thread τ_3, Fig. 4 illustrates the standard deviation $\sigma_{3,k}$, that is the square root of the expression of (21). As illustrated in the figure, $\sigma_{3,k}$ grows up to a maximum and then decreases to 0 when $k \leftarrow \mathsf{maxJ}_3$. In fact, when k grows, then the sample space over which the variance is computed shrinks. Unless the scheduling algorithm of the platform $\mathcal{P}$ can provide some level of determinism in the resource

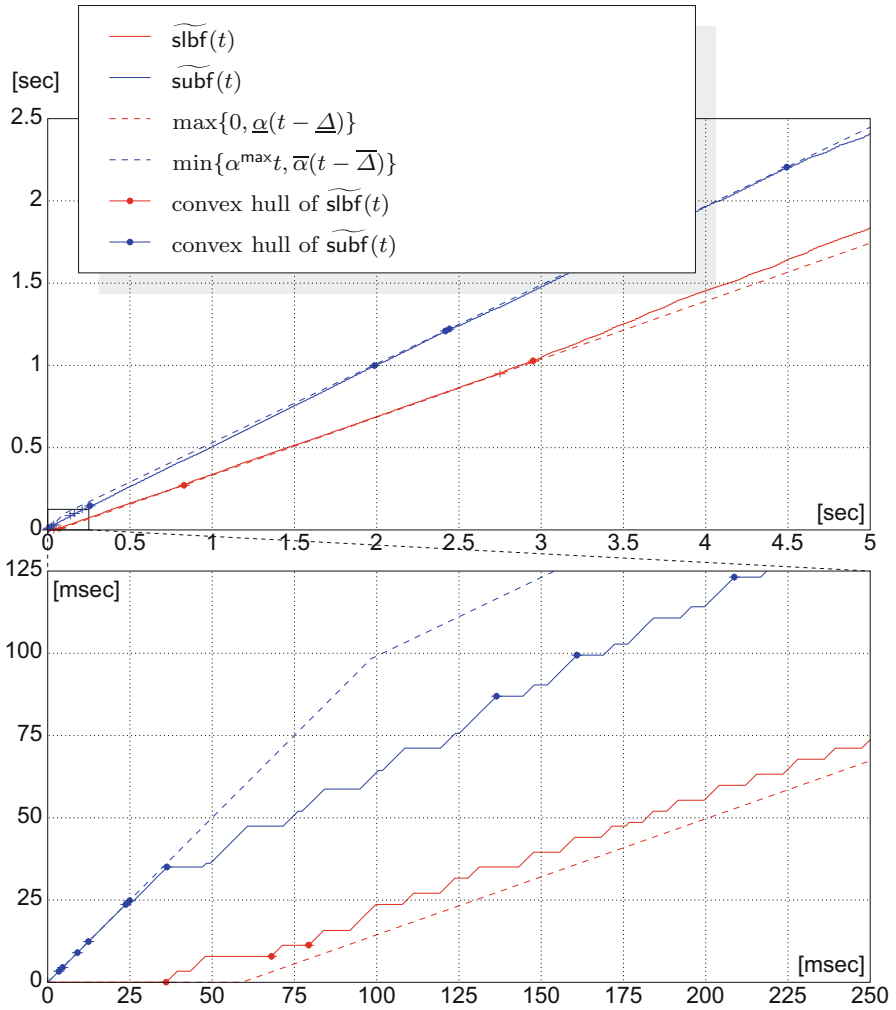


Fig. 3 Illustration of the results of the `supply` analysis module

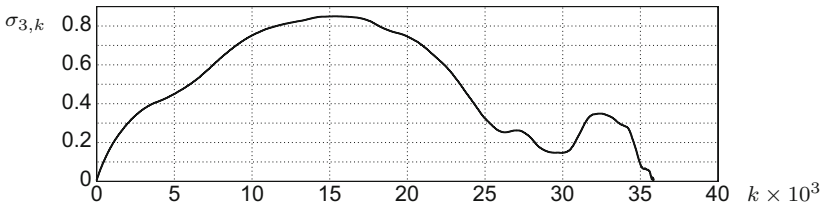


Fig. 4 Standard deviation $\sigma_{3,k}$ over k, as computed by the statistical analysis module

allocation to k consecutive jobs for some k, a decrease of $\sigma_{i,k}$ should be interpreted as an indication of a short experiment duration.

6 Experimental results

In this section, we show a set of experiments that we conducted to test the available Linux scheduling classes, as well as the capacity loss due to the usage of critical sections and memory allocation. These experiments compute the experiment-based supply function bounds $\widetilde{\mathsf{slbf}}(t)$ and $\widetilde{\mathsf{subf}}(t)$, as defined in (15)–(16), and the distribution of the threads on the available CPUs. All the experiments were run over an Intel Quad core running at 2.50 GHz equipped with Linux kernel 3.17.

6.1 Experimental environment

This section describes the experimental environment in which we performed our tests. This includes a review of the Linux scheduling hierarchy and a description of the tracing infrastructure.

In Linux, the scheduling policy is determined by the *scheduling class* assigned to a thread. The available scheduling classes in the kernel—from version 3.14—are:

1. `SCHED_DEADLINE` has the highest priority among all. It implements the Constant Bandwidth Server (CBS) (Abeni and Buttazzo 1998) and it is configured with the reservation budget Q and period P;
2. `SCHED_FIFO` implements fixed priority scheduling and is configured with a priority level from 99 (highest) to 1 (lowest);
3. `SCHED_RR` is the same as `SCHED_FIFO`. However, after a thread has run for 100 ms, (this duration is configurable by writing to `/proc/sys/kernel/sched_rr_timeslice_ms`), it is pushed back to the queue of its priority level
4. `SCHED_OTHER` this is the standard scheduling algorithm in Linux, the Completely Fair Scheduler (CFS).

Threads also have an *affinity map* which determines the set of cores where the thread is allowed to execute. For efficiency, there is one run queue per core. Hence, to implement a logical global run queue, the so-called *push/pull migration* is implemented: in certain circumstances, such as when the local queue is empty, a processor may try to pull a thread ready for execution from another queue. Similar mechanisms exist for pushing threads to other CPUs.

6.2 The `SCHED_OTHER` Linux scheduling class

The aim of this experiment is to evaluate the behavior of `SCHED_OTHER`. Despite not being a real-time scheduler, `SCHED_OTHER` is designed to enforce fairness among the threads (CFS). We defined a set $\mathcal{T} = \{\tau_1, \ldots, \tau_6\}$ of 6 threads. All the τ_i threads have the same characteristics. Their job is composed by one single phase $\phi_{i,1} = \phi^{\mathsf{compute}}(100000)$. In other words, every thread executes 100000 mathematical operations for each job. We execute the set $\mathcal{T}$ on a platform $\mathcal{P}$, composed by four cores. The scheduling parameters of τ of all the threads belonging to $\mathcal{T}$ are the same: the threads affinity mask contains three CPUs, [1, 2, 3], avoiding the execution on CPU #0. The default policy `SCHED_OTHER` is used for all the threads. We let the operating

Table 1 Threads scheduled by `SCHED_OTHER`

	`supply`				`runmap`		
τ_i	$\underline{\alpha}_i$	$\underline{\Delta}_i$	$\overline{\alpha}_i$	$\overline{\Delta}_i$	#1	#2	#3
τ_1	0.374	0.112	0.484	−0.081	0.062	0.614	0.324
τ_2	0.382	0.197	0.627	−0.313	0.191	0.332	0.476
τ_3	0.353	0.059	0.479	−0.107	0.009	0.440	0.551
τ_4	0.378	0.212	0.624	−0.319	0.159	0.302	0.539
τ_5	0.393	0.129	0.642	−0.097	0.109	0.545	0.346
τ_6	0.372	0.166	0.679	−0.181	0.834	0.062	0.104

system schedule the threads for a duration of 100 seconds. We expect that six threads with the same characteristics, running on three cores, would receive similar budgets, possibly obtaining each half of a CPU.

For each thread, Table 1 reports the pairs $(\underline{\alpha}_i, \underline{\Delta}_i)$ and $(\overline{\alpha}_i, \overline{\Delta}_i)$ computed by the `supply` module, as explained in 3.1, corresponding to the linear bounds maximizing/minimizing the area over the interval $[0, 5]$. The last three columns correspond to the share of consumed CPU obtained with the `runmap` module.

It can be observed that the allocated bandwidths are almost all equal to each other and that the range of variability of the bandwidth over time is wide as $\underline{\alpha}_i$ is always significantly smaller than $\overline{\alpha}_i$. Although other interfering operating systems threads could be easily accommodated on CPU #0, which is not used by the set $\mathcal{T}$, still the overall delivered bandwidth is lower bounded by $\underline{\alpha}_* = 2.582374$, which is quite less than the full 3 CPUs dedicated to $\mathcal{T}$. One could suspect that the missing bandwidth is due to the tracing infrastructure. However, this is not the case. As shown in the experiments of 6.3, the overhead introduced by `trace-cmd` is extremely limited and other scheduling policies manage to effectively distribute a higher fraction of the available computing resources. The supply functions of τ_3 was also plotted in 3.

Finally, by observing that the values of $\underline{\Delta}_i$ are considerably large, we conclude that the granularity of threads schedules is considerable large. This, however, is expected since `SCHED_OTHER` does not aim at providing tight timing guarantees, but rather a fair allocation of resources.

6.3 The `SCHED_FIFO` and `SCHED_RR` classes

The aim of this set of experiments is to verify the computing capacity offered by `SCHED_FIFO` and `SCHED_RR`. We expect them to offer better guarantees on the delivered computing capacity compared to `SCHED_OTHER`. In our experiments all threads run at the same priority level, which was set to 50. Otherwise, threads at higher priority would consume all the available computing capacity, and lower priority ones would never have a chance to execute.

In the first experiment, the set $\mathcal{T}$ contains seven threads, $\mathcal{T} = \{\tau_1, \ldots, \tau_7\}$. The job of each thread τ_i is composed by a single phase $\phi_{i,1} = \phi^{\mathsf{compute}}(100000)$. The affinity set of all seven threads is set to $[1, 2, 3]$, avoiding the execution on CPU #0.

Table 2 Threads scheduled by SCHED_FIFO and SCHED_RR

τ_i	scheduling class	supply $\underline{\alpha}_i$	supply $\underline{\Delta}_i$	runmap #1	runmap #2	runmap #3
τ_1	SCHED_FIFO	0.998	0.0011	0	0	1
τ_2	SCHED_FIFO	0.998	0.0003	1	0	0
τ_3	SCHED_RR	0	–	0	0	1
τ_4	SCHED_RR	0.499	0.1021	0	1	0
τ_5	SCHED_RR	0	–	1	0	0
τ_6	SCHED_RR	0.499	0.1015	0	1	0
τ_7	SCHED_RR	0	–	1	0	0

The scheduling class SCHED_FIFO is assigned to τ_1 and τ_2, while all threads in $\{\tau_3, \ldots, \tau_7\}$ are scheduled by SCHED_RR.

Given the scheduling hierarchy, we expect that the two threads executing with SCHED_FIFO would saturate one CPU each. In fact, when one of these executes for the first time on one CPU, it never releases it. The other 5 threads scheduled by SCHED_RR should all run over the third CPU, receiving then around 20% of the available computing capacity.

Table 2 shows the bandwidth $\underline{\alpha}_i$ and the delay $\underline{\Delta}_i$, extracted by the supply module. The values of $\overline{\alpha}_i$ and $\overline{\Delta}_i$ are almost identical to their corresponding lower bounds. The table also reports the percentage of time that each thread spent on each of the available CPUs, obtained with the runmap module. Contrary to our expectations, the threads scheduled with SCHED_RR, despite having the same priority level as the SCHED_FIFO ones, are not migrated to the available CPU. Hence, the SCHED_FIFO threads prevent some of them from running at all, with a quite evident lack of fairness. More precisely, after a deeper investigation, we found that τ_5 and τ_7 did not run at all. τ_3 run for a round only. Since τ_3, τ_5, and τ_7 were on the same CPU as some SCHED_FIFO thread, as the SCHED_FIFO threads starts running they never give the control back to the scheduler. In this situation, for fairness, we would expect that the scheduler would pull τ_3, τ_5, and τ_7 to the run queue of CPU #2. This, however, does not happen. The explanation is that push/pull migration happens based on the priority level only. Hence, the scheduler does not see any reason why pulling τ_3, τ_5, and τ_7 to CPU #2, when on CPU #2 it is running a thread with the same priority. We believe, however, that fairness would increase by pulling SCHED_RR threads which may have happened to be behind to some SCHED_FIFO ones in the same priority queue.[7]

Also, using the runmap module, we observe that τ_4 and τ_7 equally share CPU #2, and $\underline{\Delta}_4 = \underline{\Delta}_6 \approx 100$ ms, that is the default length of the round in SCHED_RR, as specified in /proc/sys/kernel/sched_rr_timeslice_ms. The overall bandwidth is $\underline{\alpha}_* = 2.994$, very close to the number of assigned cores. This confirms that the tracing infrastructure of trace-cmd[8] is lightweight.

[7] https://lkml.org/lkml/2015/12/11/481.

[8] https://lwn.net/Articles/410200/.

Table 3 Threads scheduled by SCHED_RR

	supply			runmap		
τ_i	$\underline{\alpha}_i$	$\underline{\Delta}_i$	$\overline{\Delta}_i$	#1	#2	#3
τ_1	0.499088	0.103804	−0.101733	0	1	0
τ_2	0.332735	0.201808	−0.202646	1	0	0
τ_3	0.332756	0.202721	−0.202454	1	0	0
τ_4	0.499114	0.102200	−0.197099	0	0	1
τ_5	0.499065	0.104840	−0.101514	0	1	0
τ_6	0.332763	0.202703	−0.202447	1	0	0
τ_7	0.499093	0.197762	−0.102400	0	0	1

In a follow-up experiment with SCHED_RR the same 7 threads were all assigned the SCHED_RR scheduling class. As shown in Table 3, despite the threads having all the same characteristics, the resource allocation is uneven. This happens because the scheduler does not migrate threads to achieve fairness. In fact, the bandwidth of each thread is equally divided among threads running over the same CPU.

6.4 Impact of memory allocation

The aim of this set of experiments is to quantify the overhead of dynamically allocating memory to the running threads. We run a single thread $\mathcal{T} = \{\tau_1\}$. The thread is composed by a single phase $\phi_{i,1} = \phi^{\text{memory}}(1000000, m)$, with m being the size of an array of double precision floating-point variables dynamically allocated/deallocated by the thread (see Sect. 4). In the experiment, we set $m \in \{[1, 2, 3, 4, 5, 6, 7, 8, 9] \times 10^d\}$ and $d \in \{0, 1, 2, 3, 4, 5, 6\}$. Keeping a constant number of operations and by increasing the size of the memory used by the thread, we aim at evaluating the impact of allocating and freeing memory as well as the cache size. The processors of our test machine have 128 KB of L1 cache and 4096 KB of L2 cache. The size of a double number is 8 bytes, therefore our tests use a range of memory spanning from 8 B ($m = 1$) to 72 MB ($m = 9000000$).

We use SCHED_RR and set τ_1's affinity mask to [1, 2, 3], avoiding the execution on CPU #0. Also, we set the experiment duration equal to 300 seconds, to evaluate the effect over a longer run. We report the average job duration and the results obtained with the migration module. The first row in 5 shows the average jobs duration for different memory sizes. As can be seen, the average value spikes up, and the job becomes on average twice as long, in the transition between $m = 4000000$ and $m = 5000000$.

The second plot in Fig. 5 shows the migration ratio, defined as the amount of migrations over the amount of completed jobs. As can be seen, the ratio is quite low until the jobs have a short duration. Once the jobs have longer duration, the ratio of migration increases, showing that the scheduler is actively trying to cope with the new job, possibly hoping that migrating the thread would help in reducing its duration.

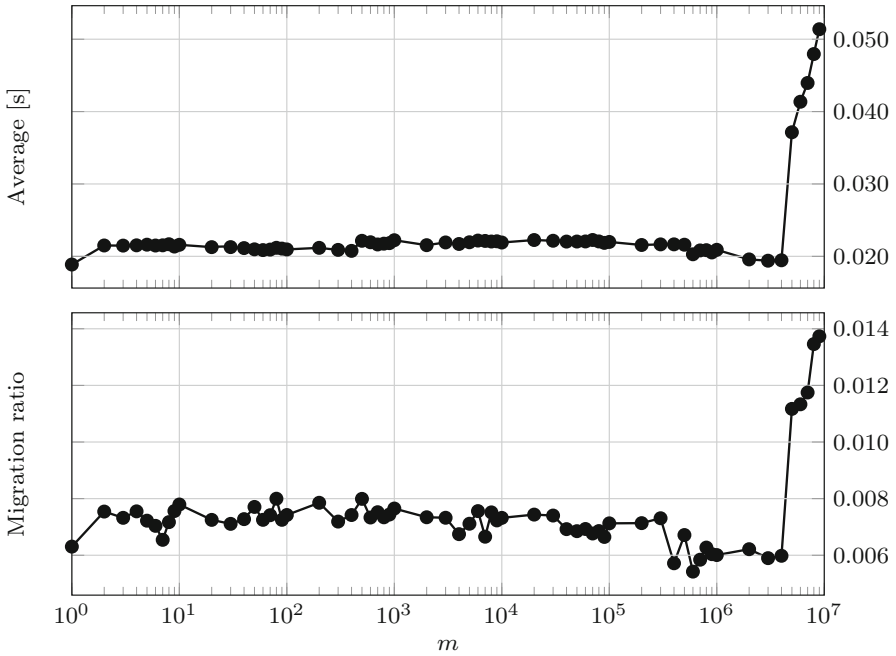


Fig. 5 Mean job duration and migration ratio (# of migrations divided by the # of completed jobs) varying the amount of memory used in each job

6.5 The `SCHED_DEADLINE` scheduling class

The aim of this set of experiments is to investigate the behavior of the highest priority Linux scheduling policy, `SCHED_DEADLINE`. `SCHED_DEADLINE` in principle should offer the best timing guarantees among all the policies implemented in the Linux kernel, since it has been specifically designed for the real-time execution of tasks. In the following, we test its behavior both in terms of it enforcing the parameters of a single thread in isolation and in terms of the amount of computation given to a thread with more complex tasksets.

First, we report a series of tests with one single thread τ_1, varying the scheduling parameters. Specifically, we varied the budget $\tau_1.Q$ and correspondingly the period $\tau_1.P$ of the thread, keeping the ratio between them fixed and equal to 0.5. As done with previous experiments, the affinity mask of the thread contains three cores, [1, 2, 3], while core 0 was kept free. The thread τ_1 is composed by a single compute phase $\phi_{i,1} = \phi^{\mathsf{compute}}(10000)$.

The x-axis of Fig. 6 shows $\tau_1.Q$ while the y-axis hosts the figures $\underline{\alpha}_1$ and $\underline{\Delta}_1$. As can be seen, the use of `SCHED_DEADLINE` results in very predictable lower bounds. With very short periods (less than 1ms), $\underline{\alpha}_1$ is close to 0.45, while higher budgets result in a better use of the available resource, with $\underline{\alpha}_1$ approaching the theoretical value of 0.5. When even higher budgets are given to the τ_1, despite having a lower bound $\underline{\alpha}_1$ very close to 0.5, $\underline{\Delta}_1$ increases, reaching values corresponding to 0.1 second.

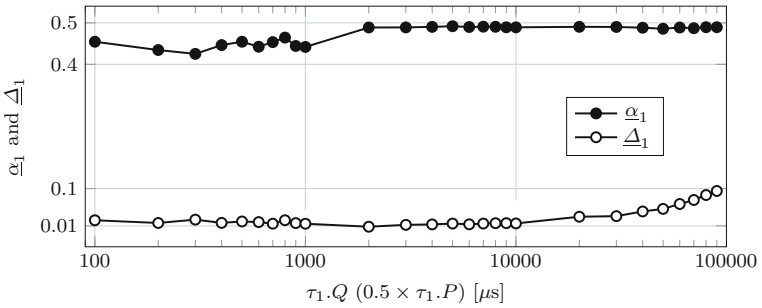


Fig. 6 $\underline{\alpha}_1$ and $\underline{\Delta}_1$ obtained with different couples of budget Q and period P and a fixed ratio of 0.5

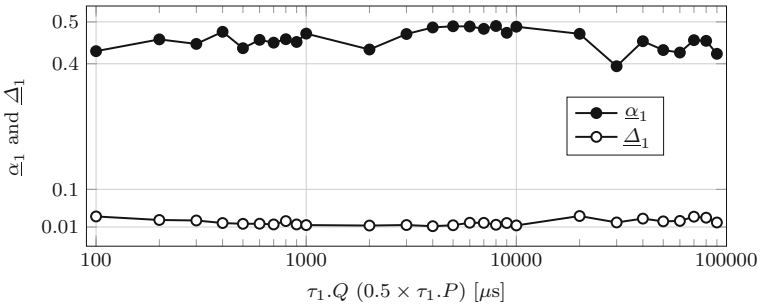


Fig. 7 $\underline{\alpha}_1$ and $\underline{\Delta}_1$ obtained with different couples of budget Q and period P, and a fixed ratio of 0.5, with additional load

While this test experimentally demonstrated the ability of `SCHED_DEADLINE` to provide accurate resources to the running threads, it is worth investigating how it behaves in higher load conditions. We therefore tested the limits of `SCHED_DEADLINE` in a situation close to overload. We used a set of threads $\mathcal{T} = \{\tau_1, \tau_2, \tau_3, \tau_4\}$. All the four threads can be scheduled on two of the four available CPUs, its affinity mask being [1, 2]. We conducted a series of experiment, varying the budget and the period of τ_1 as done for the previous experiment. The other threads, $\tau_{\{2..4\}}$ receive a fixed budget of 10 ms with a period of 20 ms. All the four threads are composed by a single phase $\phi_{i,*} = \phi^{\mathsf{compute}}(10000)$.

Figure 7 shows $\underline{\alpha}_1$ and $\underline{\Delta}_1$ for each of the tested values. The behavior obtained with a budget up to 10 ms is very similar to the one shown in Fig. 6. On the contrary, when the budget and period increase, $\underline{\alpha}_1$ starts decreasing, becoming less predictable. This is not surprising, considering the stress introduced in the system with the additional load.

In another experiment, we aimed at plotting the experiment-based supply lower and upper bounds to the resource provided by a `SCHED_DEADLINE` thread. Figure 8 shows the supply function bounds of a thread scheduled

1. within a reservation with budget $Q = 10\mathsf{msec}$ and period $P = 20\mathsf{msec}$ (in black), and

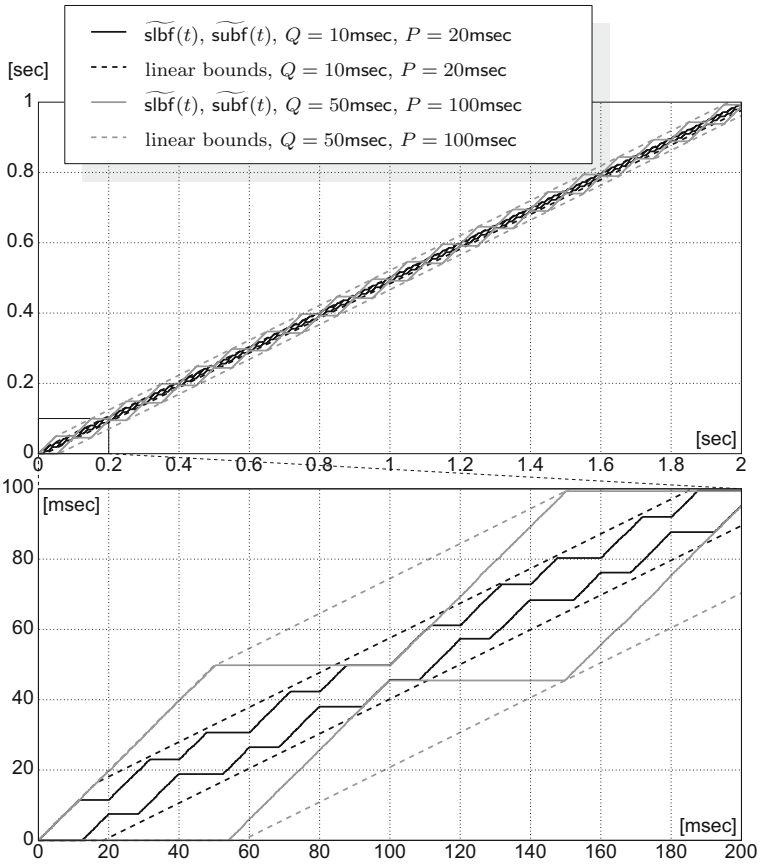


Fig. 8 Supply functions of a `SCHED_DEADLINE` thread

2. within a reservation with budget $Q = 50\mathsf{msec}$ and period $P = 100\mathsf{msec}$ (in gray).

Also, the linear bounds are drawn by dashed lines. In addition to the analized thread, 7 more threads create load to the 4 CPUs, such that the total load by the 8 threads to the 4 CPUs is about 3.5.

In the case of period $P = 20\mathsf{msec}$, the lower bound to the bandwidth is $\underline{\alpha}_1 = 0.495127$, very close to the theoretical value of 0.5. Similarly, in the case of period $P = 100\mathsf{msec}$, the lower bound to the bandwidth is $\alpha_1 = 0.495218$.

6.6 Decrease of capacity with shared resources

The aim of this set of experiments is to investigate the variation in the total offered computing capacity $\underline{\alpha}_*$ due to critical sections. To do so, we define a set of n threads $\mathcal{T} = \{\tau_1, \ldots, \tau_n\}$ with n belonging to the set $\{3, 5, 10\}$. All jobs of all threads are composed by two different phases: $\phi_{i,1} = \phi^{\mathsf{lock}}(x \times 10000, 0)$ and

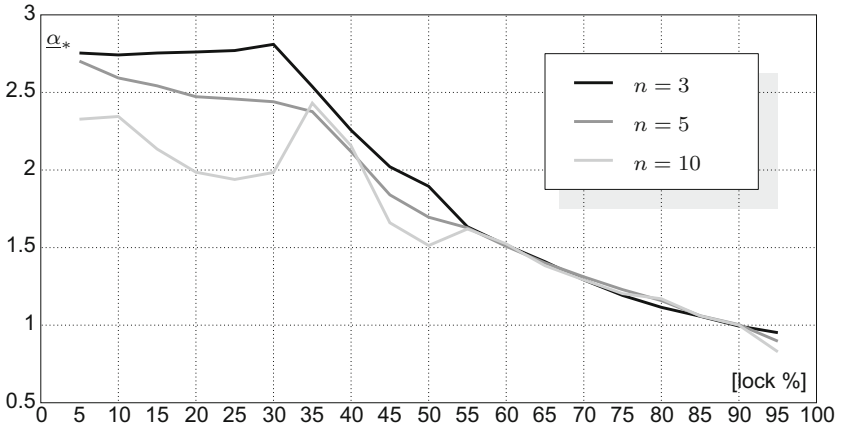


Fig. 9 Total computing capacity $\underline{\alpha}_*$ as the size of critical section varies

$\phi_{i,2} = \phi^{\text{compute}}((1 - x) \times 10000)$. Hence, a fraction x of the total 10000 mathematical operations are executed while locking the resource "0". The rest is executed outside the locked section. The experiment is repeated with the fraction x of operations within the locked section ranging over $\{5\%, 10\%, \ldots, 95\%\}$. The affinity mask of all threads is set to [1, 2, 3], avoiding the execution on CPU #0. All the n threads execute within a CBS server, implemented by the `SCHED_DEADLINE` scheduling class, with period $P = 10$ ms and budget $Q = \frac{3}{n} \times 10$ ms, with n being the total number of threads. With these settings, all n threads equally share the 3 available cores. We are interested in measuring the total computing capacity provided by the platform, $\underline{\alpha}_*$. We expect that an increase in the number of threads would generate more contention on shared resources and, at the same time, an increase in the size of the critical section would diminish the provided total capacity $\underline{\alpha}_*$.

Figure 9 shows the total computing capacity $\underline{\alpha}_*$. The most surprising, and to some extent unexpected, behavior is the case with $n = 10$ threads. When the fraction of locked code x is around 35% the delivered computing capacity is actually at its maximum (even higher than when $x = 5\%$). We have found a sound explanation. When the share of locked section x is larger than $1/m$, with m equal to the number of CPUs, the thread schedule is constrained by the schedule of the locked sections. This actually simplifies greatly the scheduling decisions and the resulting capacity $\underline{\alpha}_*$ is higher. If instead $x < 1/m$ (33% in our case) then the scheduler has to take decisions over n threads, with a clear performance degradation proportional to the number of threads. Considering the counter-intuitive nature of this phenomenon, we plan further investigations in the future. Finally, we also observe that for very large fraction of locked job body, the overall computing capacity is even less that a single core. This is not surprising, since under these conditions much time is wasted by managing the locks.

Along the same line, we construct another experiment, varying the number of threads and fixing three different percentage of operations in the critical section: 25, 50 and 75%. For each thread, the period of the reservation P is set to 10000 (μs)

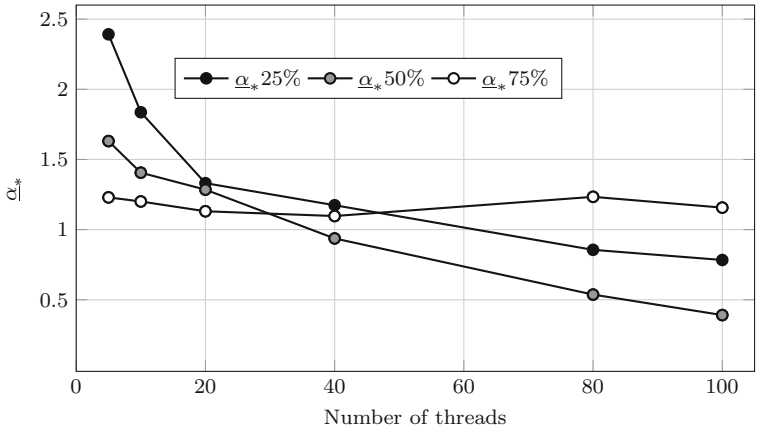


Fig. 10 $\underline{\alpha}_*$ with a varying number of threads and different relative critical sections sizes

and the budget $Q = 3 \times 10000/n$ where n is the total number of threads. With these parameters, threads equally share the 3 available cores. We are interested in measuring the lower bound on the total computing capacity provided by the platform, $\underline{\alpha}_*$. We expect that an increase in the number of threads would generate more contention on shared resources and, at the same time, an increase in the size of the critical section would diminish the provided total capacity.

Figure 10 shows three lines, each of them corresponding to the relative size of the critical section (25, 50 and 75%) with respect to the number of threads displayed on the x-axis. For short critical sections (25 and 50%), the lower bound on the total delivered capacity $\underline{\alpha}_*$ is diminishing sharply with the number of threads, with a provided capacity even below 1 when the number of threads is about 60 and 40, respectively. However, when the critical section takes up 75% of the total work done by the job of the threads, the total capacity is always above 1, even when the number of threads increases up to 100.

7 Related work

Experimental results are crucial in the development of software systems and schedulers are no exceptions (Chodrow et al. 1991). Indeed, there are many tools that aim at testing and verifying different aspects of schedulers for real-time systems. Some of them executes specific programs to test one of the capabilities of the real-time system, like memory management (Chodrow et al. 2006), mutexes and priority inheritance,[9] resource holding times (Fisher et al. 2007), or scheduling with the control groups.[10] Other works are devoted to specific type of applications, like quantifying the gap between models and implementations of control systems (Yazarel et al. 2005) or the behavior of multimedia applications (Li et al. 2006). `rt-muse` is different with

[9] https://github.com/linux-test-project/ltp.

[10] http://sourceforge.net/projects/cgfreak/.

respect to both these classes, since it extracts performance metrics running a generic program, that represents the real application in its entirety. This is not the first work using measurements from a real system to extract characteristics and models of different aspects of the execution of a program on a particular architecture. A notable example is (Baier et al. 2015), where the authors use measurement-based techniques to fine-tune stochastic models that can be analyzed by a probabilistic model checker, applying the technique to the analysis of a test-and-set spinlock protocol (Anderson 1990). Despite the similarity of performing tests on a real architecture to extract models, the models and the aim of the two projects are complementary. The main idea of (Baier et al. 2015) is to derive models from current architectures that could be extended to provide insight on the behavior of future platforms. However, it is quite cumbersome to obtain precise and reliable information about the operating systems behavior (Traeger et al. 2008). Due to this, we are satisfied with approximate computation models and the generated upper and lower bounds.

Other tools like Hourglass (Regehr 2002) and `rt-muse` execute a specific workload that could be specified as a command line parameter. Hourglass aims at testing the behavior of schedulers for uniprocessor platforms. It computes a schedule for a given workload, by executing the workload and recording which thread was running and when context switches were happening. While `rt-muse` has in common with Hourglass the execution of a workload that could be customized, `rt-muse` targets multicore platforms and provides experimental bounds to the amount of CPU provided in the given run. Since Hourglass provides a schedule, it is quite difficult to quantify the results and understand if something could be improved in the implementation of the scheduler. One of the earliest tools for analyzing schedulers was the Hartstone benchmark (Kamenoff and Weiderman 1991). Hartstone aims at testing the capabilities of a uniprocessor architecture by executing tests of increasing difficulty. In every iteration, the pool of threads to be scheduled generates more load for the architecture and the number of iterations that could be executed without deadline misses determines the benchmark score. While being extremely simple to implement in every architecture, Hartstone lacks generality, since it assumes that threads do not interfere with each other, neither with shared memory nor with messages or similar. It also does not provide a clear interpretation of the results.

In an attempt to measure the effect of the most known sources of bottlenecks in production environments, `lmbench` (McVoy and Staelin 1996) provides a set of microbenchmarks to test different performance issue sources in modern operating systems. Similarly to `rt-muse`, it measures latencies and delays. However, the intent of `lmbench` is to reproduce problems and issues and not to benchmark the implementation of real-time environments. The benchmark is not application-independent, nor it provides further analysis capabilities on the obtained result for the platform as a whole. A widely used real-time benchmark, cyclictest[11] is a high-resolution analysis tool to test a real-time system in its entirety, including the architecture, the scheduler and the workload. It measures the latency response of an application that only sleeps by continuously setting the instant in which the application should be woken up and

[11] http://www.kernel.org/pub/linux/kernel/people/clrkwllms/rt-tests/.

records the difference between this instant and the real wake up time. It is useful to detect unexpected large latencies on a system, for example due to overload conditions. This benchmark is very difficult to extend and does neither provide any insight on the behavior of the application in case its load is complex, nor combine different types of operations. The suite of MiBench (Guthaus et al. 2001) is composed by 35 applications that could be used in the context of embedded and real-time systems, for example in telecommunications or in the automotive domain. Despite it being a complete workload, it does not provide any data analysis.

Bastoni et al. (2011) studied semi-partitioned schedulers in depth, evaluating the scheduling overhead, the cache-induced delays and other metrics with LITMUS^{RT} (Calandrino et al. 2006). While both `rt-muse` and (Bastoni et al. 2011) use traces of events to analyze the behavior of schedulers, LITMUS^{RT} aims at simplifying the implementation of complex scheduling policies, while `rt-muse` aims at closing the gap between theory and implementation by exposing the real-time characteristics of already implemented schedulers.

8 Conclusions and future work

To respond to the need of profiling the real-time behavior of real-time schedulers, we proposed `rt-muse`. Due to the extensible nature of the tool, we plan to implement new phases which uses disk, network, or simply sleep. In addition, the results of the analysis could be used to propose some patches in the Linux kernel, such as the introduction of more fairness when scheduling together `SCHED_FIFO` and `SCHED_RR` threads.

Acknowledgements The authors would like to thank Tommaso Cucinotta for his insightful comments on an earlier version of this draft.

References

Abeni L, Buttazzo G (1998) Integrating multimedia applications in hard real-time systems. In: Proceedings of the 19th IEEE real-time systems symposium, pp 4–13

Åkesson B, Goossens K (2011) Architectures and modeling of predictable memory controllers for improved system integration. In: Design, automation & test in europe conference & exhibition, pp 1–6

Almeida L, Pedreiras P, Fonseca JAG (2002) The FTT-CAN protocol: why and how. IEEE Trans Ind Electron 49(6):1189–1201

Anderson JH, Srinivasan A (2001) Mixed Pfair/ERfair scheduling of asynchronous periodic tasks. In: Proceedings of the 13th Euromicro conference on real-time systems, pp 76–85

Anderson TE (1990) The performance of spin lock alternatives for shared-memory multiprocessors. IEEE Trans Parallel Distrib Syst 1(1):6–16

Baccelli F, Cohen G, Olsder GJ, Quadrat JP (1992) Synchronization and linearity, vol 3. Wiley, New York

Baier C, Daum M, Engel B, Härtig H, Klein J, Klüppelholz S, Märcker S, Tews H, Völp M (2015) Locks: picking key methods for a scalable quantitative analysis. J Comput Syst Sci 81(1):258–287

Bastoni A, Brandenburg BB, Anderson JH (2011) Is semi-partitioned scheduling practical? In: Proceedings of the 2011 23rd Euromicro conference on real-time systems, pp 125–135

Bini E, Bertogna M, Baruah S (2009) Virtual multiprocessor platforms: specification and use. In: Proceedings of the 2009 30th IEEE real-time systems symposium, pp 437–446

Brandenburg B, Anderson J (2007) Feather-trace: a light-weight event tracing toolkit. In: Proceedings of the third international workshop on operating systems platforms for embedded real-time applications, pp 19–28

Calandrino JM, Leontyev H, Block A, Devi UC, Anderson JH (2006) Litmus^{RT}: a testbed for empirically comparing real-time multiprocessor schedulers. In: Proceedings of the 27th IEEE international real-time systems symposium, pp 111–126

Chodrow S, Jahanian F, Donner M (1991) Run-time monitoring of real-time systems. In: Proceedings of the 12th real-time systems symposium, pp 74–83

Cruz RL (1991) A calculus for network delay, part I: network elements in isolation. IEEE Trans Inf Theory 37(1):114–131

Fisher N, Bertogna M, Baruah S (2007) Resource-locking durations in EDF-scheduled systems. In: Proceedings of the 13th IEEE real time and embedded technology and applications symposium, pp 91–100

Guthaus MR, Ringenberg JS, Ernst D, Austin TM, Mudge T, Brown RB (2001) MiBench: a free, commercially representative embedded benchmark suite. In: 2001 IEEE international workshop on proceedings of the workload characterization. WWC-4, pp 3–14

Kamenoff N, Weiderman N (1991) Hartstone distributed benchmark: requirements and definitions. In: Proceedings of the twelfth real-time systems symposium, pp 199–208

Li ML, Van Achteren T, Brockmeyer E, Catthoor F (2006) Statistical performance analysis and estimation of coarse grain parallel multimedia processing system. In: Proceedings of the 12th IEEE real-time and embedded technology and applications symposium, pp 277–288

Lipari G, Bini E (2003) Resource partitioning among real-time applications. In: 15th Euromicro conference on real-time systems, pp 151–161

Maggio M, Lelli J, Bini E (2016) A tool for measuring supply functions of execution platforms. In: 2016 IEEE 22nd international conference on embedded and real-time computing systems and applications (RTCSA), pp 39–48, Aug 2016

McVoy L, Staelin C (1996) LMBench: portable tools for performance analysis. In: Proceedings of the 1996 annual conference on USENIX annual technical conference, pp 23–23

Mok AK, Feng X, Chen D (2001) Resource partition for real-time systems. In: Proceedings of the 7th IEEE real-time technology and applications symposium, pp 75–84

Pellizzoni R, Bui BD, Caccamo M, Sha L (2008) Coscheduling of CPU and I/O transactions in COTS-based embedded systems. In: Proceedings of the 2008 real-time systems symposium, pp 221–231

Pizlo F, Vitek J (2006) An emprical evaluation of memory management alternatives for real-time java. In: Proceedings of the 27th IEEE international real-time systems symposium, pp 35–46

Regehr J (2002) Inferring scheduling behavior with hourglass. In: Proceedings of the FREENIX track: 2002 USENIX annual technical conference, pp 143–156

Regnier P, Lima G, Massa E, Levin G, Brandt S (2011) RUN: optimal multiprocessor real-time scheduling via reduction to uniprocessor. In: Proceedings of the 32nd IEEE real-time systems symposium, pp 104–115, Dec 2011

Shin I, Lee I (2003) Periodic resource model for compositional real-time guarantees. In: Proceedings of the 24th IEEE international real-time systems symposium, pp 2–13

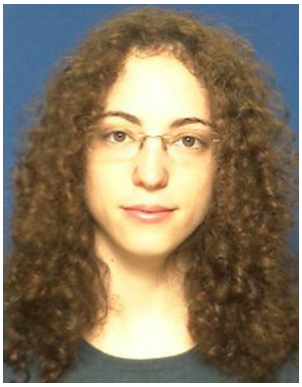
Thiele L, Chakraborty S, Naedele M (2000) Real-time calculus for scheduling hard real-time systems. In: Proceedings of the IEEE international symposium on circuits and systems, pp 101–104

Traeger A, Zadok E, Joukov N, Wright CP (2008) A nine year study of file system and storage benchmarking. Trans Storage 4(2):5:1–5:56

Xu M, Phan LT, Sokolsky O, Xi S, Lu C, Gill C, Lee I (2015) Cache-aware compositional analysis of real-time multicore virtualization platforms. Real Time Syst 51(6):675–723

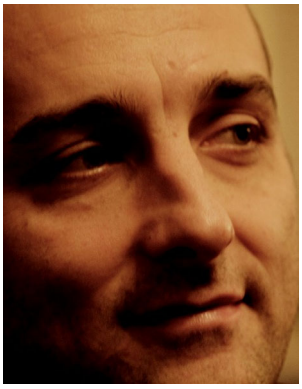
Yazarel H, Girard A, Pappas GJ, Alur R (2005) Quantifying the gap between embedded control models and time-triggered implementations. In: Proceedings of the 26th IEEE international real-time systems symposium, pp 111–120

Yun H, Yao G, Pellizzoni R, Caccamo M, Sha L (2016) Memory bandwidth management for efficient performance isolation in multi-core platforms. IEEE Trans Comput 65(2):562–576



Martina Maggio is an Associate Professor at the department of Automatic Control, Lund University. The focus of her PhD studies at the Dipartimento di Elettronica e Informazione at Politecnico di Milano, supervised by Alberto Leva, was the control-theoretical design of computing systems components. For one year, she visited the Computer Science and Artificial Intelligence Laboratory, at Massachusetts Institute of Technology, working under the supervision of Anant Agarwal and together with Henry Hoffmann on the Self-Aware Computing project, named one of ten "World Changing Ideas" by Scientific American in 2011. During her stay at the Lund University, she worked with Karl-Erik Årzén on resource management in Real-Time computing systems and cloud computing problems. Her research interests include the application of control theory to Software Engineering problems, with the goal of designing software systems that provide predictable performance despite runtime variations.

Juri Lelli received a BS and a MS in Computer Engineering at the University of Pisa (Italy). He then earned a PhD degree at the Scuola Superiore Sant'Anna of Pisa, Italy (ReTiS Lab). He is one of the original authors of the SCHED_DEADLINE scheduling policy in Linux, and he is actively helping maintaining it. He is currently working at ARM Ltd., where he continues contributing to the Linux scheduler development, with a special focus on energy aware scheduling and power management.



Enrico Bini is Associate Professor at Department of Computer Science, University of Turin. Until 2016, he was assistant professor at Scuola Superiore Sant'Anna, Pisa. Also in 2012–14, he was Marie-Curie fellow at Lund University. In 2004, he completed the PhD on Real-Time Systems at Scuola Superiore Sant'Anna (recipient of the Spitali Award for best PhD thesis of the whole university). In January 2010 he also completed a Master degree in Mathematics with a thesis on optimal sampling for linear control systems. He has published more than 90 papers (4 best-paper awards @RTNS @RTCSA @ICC) on real-time scheduling, operating systems, optimization methods for real-time and control systems, optimal management of distributed and parallel resources. His service to the research community includes the participation in 58 Technical Program Committees (including RTSS (2010, 2014, 2015, 2017), RTAS (2009, 2010, 2011, 2013, 2015, 2016) and EMSOFT (2011, 2016)), the organization of 12 events (including PC co-Chair of RTNS 2017, Local co-Organizer of ESWeek 2018), the review of 9 PhD thesis and about 40 papers/year, in the above mentioned research areas.